LINE FOLLOWING BEHAVIOR FOR AN AUTONOMOUS MOBILE ROBOT USING ARTIFICIAL NEURAL NETWORKS

Contents

Abstract

In order to achieve tasks by the mobile robots, these robotic systems must have been intelligent and should decide their own action. To guarantee the autonomy and the intelligence for line following behavior, it is necessary to use the techniques of artificial intelligence like the artificial neural networks. This project report presents an approach for line following task by an autonomous mobile robot using a single layer neural network. The proposed controller is used for following any line on a plain surface with different width. This controller can also be upgraded to determine the value of k_p and k_d and make the autonomous line following a PD controller based. The results acquired from Neural Network simulation and implementation on the robot are shown and discussed.

Keywords- Feedforward Neural Network, Machine Learning, Robotics, Backpropagation Algorithm, Stochastic Gradient Descent, Proportional Derivative Controller

1 Introduction

The line follower is a self operating robot that detects and follows a line that is drawn on the floor. The path consists of a black line on a white surface (or it may be reverse of that). The control system used must sense a line and maneuver the robot to stay on course, while constantly correcting the wrong moves using feedback mechanism, thus forming a simple yet effective closed loop system. The robot is designed to follow very tight curves.

In this project, the conventional control system is being replaced by artificial neural network.

1.1 Line detection mechanism

Line can be detected by either using Infra-red (IR) sensors, Light Dependent Resistor (LDR) or by a camera with line detection algorithms. Line tracking is a very important notion in the world of robotics as it give to the robot a precise, error-less and easy to implement navigation scheme.

Detecting line using IR To detect line using IR, a threshold value must be measured. IR sensor gives different reading on different colored surface. Both values from IR on white line and IR on black line must be recorded before sensing the line.

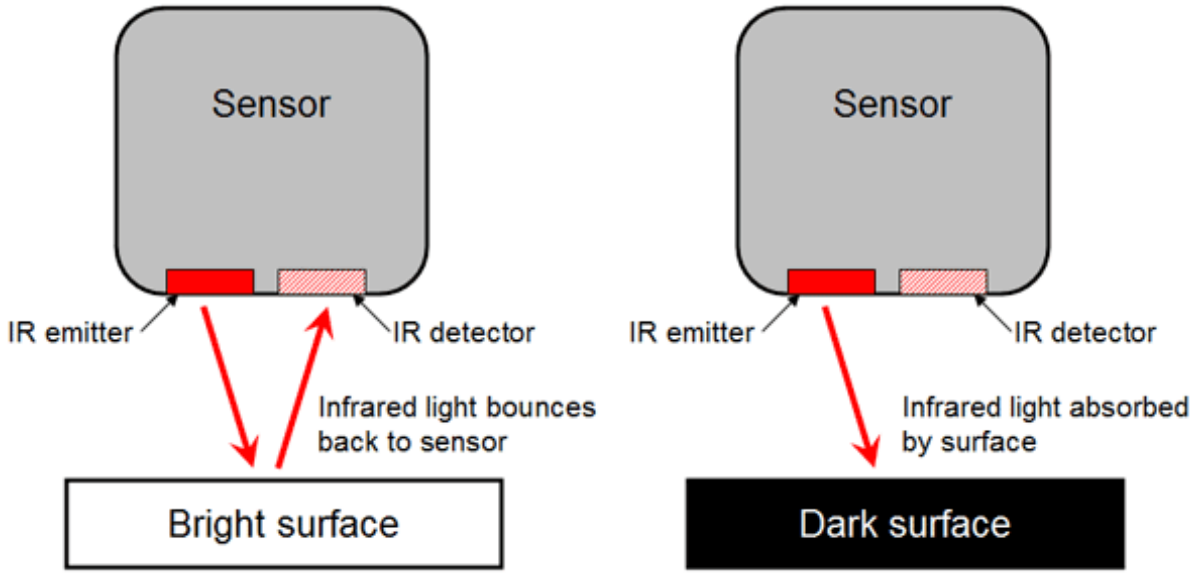


Figure 1: Detecting line using IR sensor

Detecting line using LDR To detect line using LDR same procedure from IR can be used. Since reflected line intensity depends on the reflecting surface. If the color of the surface is white, maximum light is reflected. If it is black then minimum light is reflected. Reflected light has different intensity based on the color of the surface it is being reflected from. So, nearest colors can be differentiated using LDR sensors.

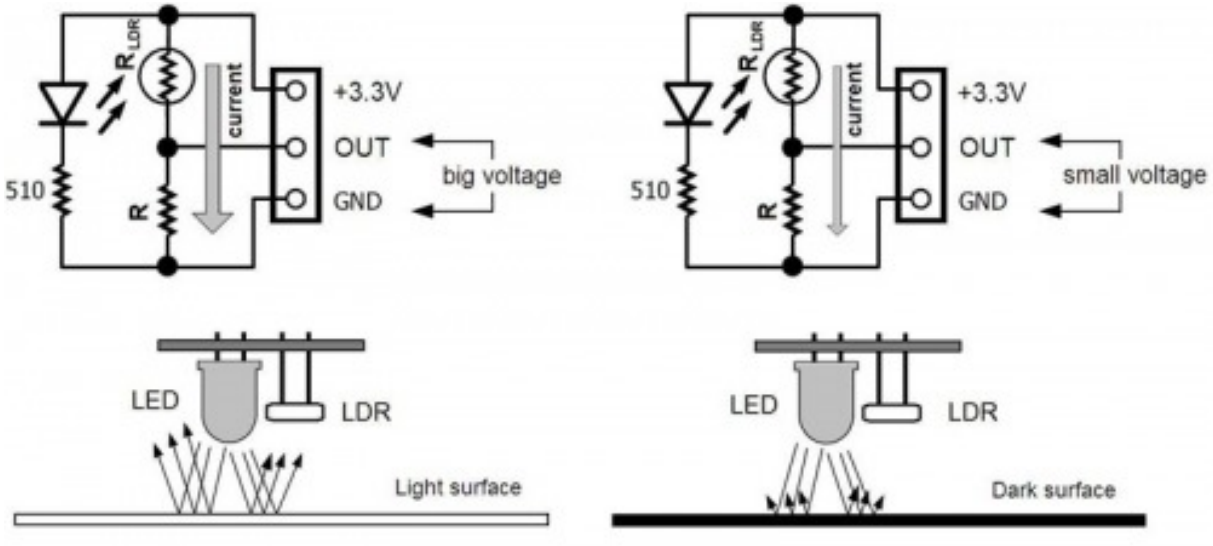


Figure 2: Detecting line using LDR sensor

1.2 Algorithm for detecting line

Algorithm 1 Line Detecting Algorithm

```
1: procedure DETECTLINE(irPin)
2:     irPin ← ir receiver pin
3:     irReading ← analog reading from ir receiver
4:     threshold ← threshold value for differentiate betweeen white and black line
5:     irDigitalReading ← converts analog into binary format
6:     irReading ← reading from irPin
7:     if irReading > threshold then
8:         irDigitalReading ← 1
9:     else
10:        irDigitalReading ← 0
```

1.3 Getting current position of the robot using IR / LDR Array

Current position of a robot on a line can be extracted by using IR array consisting of two or more than two IR/LDR sensors. Suppose, a robot has an IR array of 5 IR sensors. The spacing between two IR sensor is **1cm**, and the width of the line is **1.5cm**. At any time, when the robot is on the line, two values will differ from other three values meaning robot is facing straight, leaning left or leaning right. Depending on the position of the robot on the line, speed of motors can be varied to keep it on the line. This process is completely experimental and varies with the body, circuitry, battery rating, motor rating and mostly other things. The position value can be returned either in binary form such as **01100** using 1 or in weighted value such as **2500** from 2.

1.3.1 Algorithm for weighted positional value

Algorithm 2 Position calculating algorithm

```
1:  procedure CALCULATEPOSITION(numberOfSensors)
2:      numberOfSensors ← number of ir/ldr sensors
3:      numberOfActiveSensors ← 0
4:      digitalReading[numberOfSensors] be new array
5:      weightedValues ← 0
6:      weight ← 1000                          ▷ Setting weighted value 1000
7:      position ← −1                          ▷ Setting current position at -1
8:      for index ← 0 to numberOfSensors do
9:          digitalReading[index] ← DETECTLINE(index)
10:         if digitalReading[index] ← 1 then
11:             numberOfActiveSensors ← numberOfActiveSensors + 1
12:         weightedValues ← weightedValues + digitalReading[index] ∗
    weight
13:     position ← weightedValues/numberOfActiveSensors
14:     return position
```

1.4 Conventional line following mechanism

Conventional line following robots follow lines on a surface based on either predefined conditions or line patterns. Position of the robot can be calculated from 2. If the position of the sensor indicates the robot is being shifted to right from mid point of the line, then the speed of left motor is increased and right motor is decreased and vice versa. If the position of the robot is at middle point then both of the motor will go in the same direction with same speed. The procedure can be viewed from figure 3.

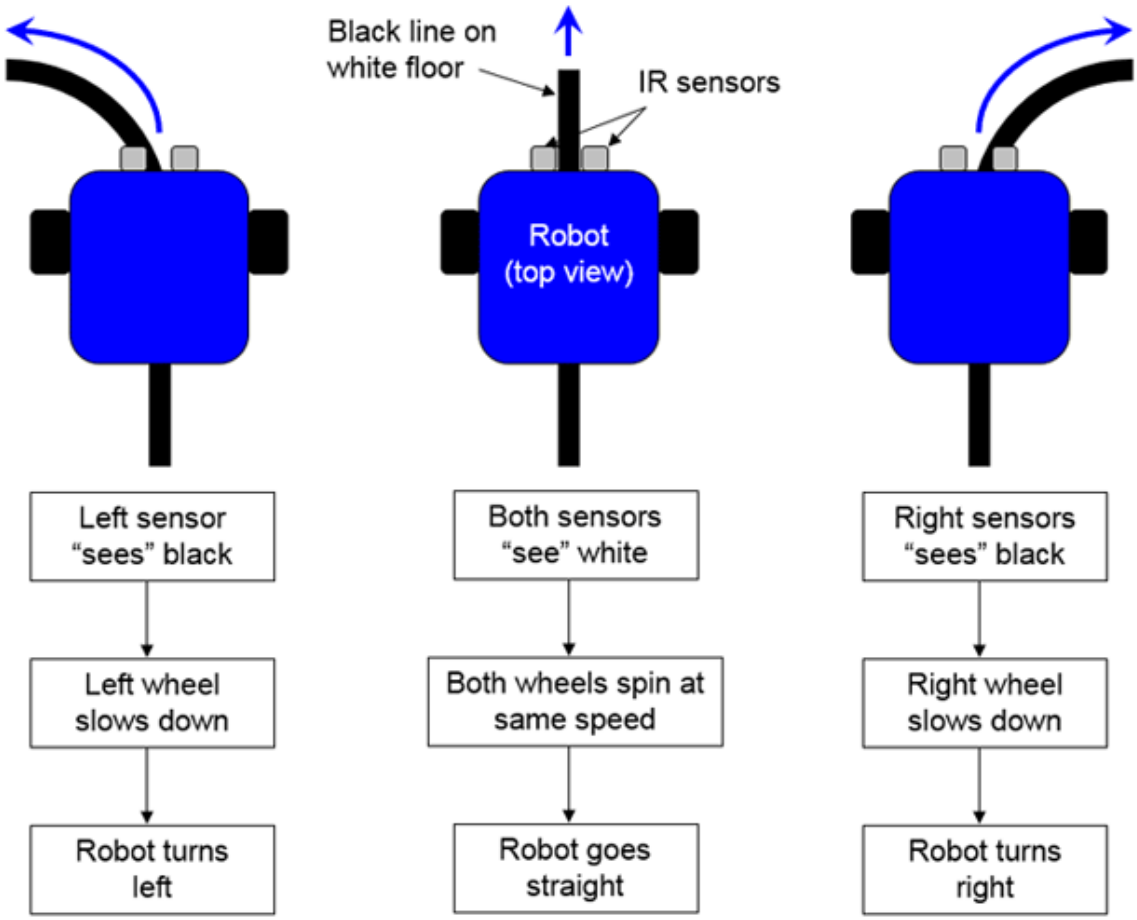


Figure 3: Differential steering drive for following line

1.5 Drawbacks of line following robot based on differential drive system

Some drawbacks of differential drive system

1. It is a static method that can only be used on a specific robot for which the algorithm was designed
2. Conditional statements varies with
   (a) Weight of the robot
   (b) Speed of the robot
   (c) Spacing between the IR/LDR sensors of the sensor array
   (d) Number of IR/LDR sensors used to make the array
   (e) Width of the line to be followed by the robot

3. This method is not appropriate for driving the robot in right and acute angle turns

2 Machine Learning

Machine learning is a subfield of computer science that evolved from the study of pattern recognition and computational learning theory in artificial intelligence. In 1959, Arthur Samuel defined machine learning as a "Field of study that gives computers the ability to learn without being explicitly programmed".

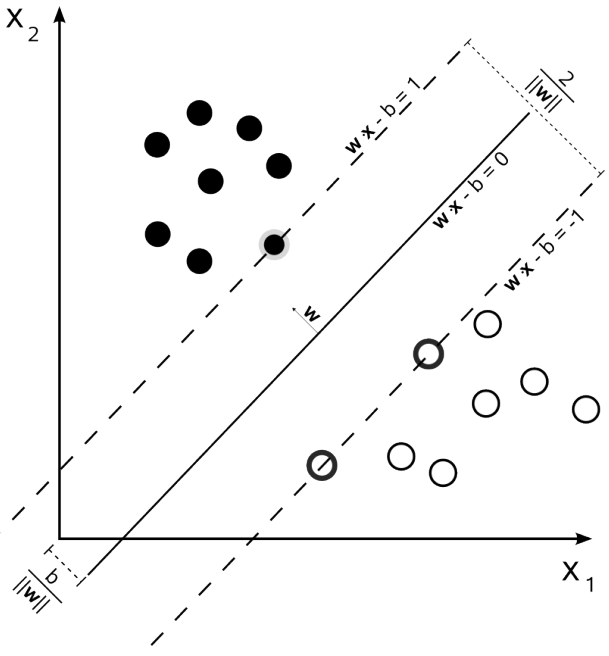


Figure 4: Support Vector Machine (SVM), an algorithm to classify data

2.1 Model of an Artificial Neuron

An artificial neuron is a mathematical function conceived as a model of biological neurons. Artificial neurons are the constitutive units in an artificial neural network. Depending on the specific model used they may be called a semi-linear unit, Nv neuron, binary neuron, linear threshold function, or McCullochPitts (MCP) neuron. The artificial neuron receives one or more inputs (representing dendrites) and sums them to produce an output (representing a neuron's axon). Usually the sums of each node are weighted, and the sum is passed through a non-linear function known as an activation function or transfer function. The transfer functions usually

have a sigmoid shape, but they may also take the form of other non-linear functions, piecewise linear functions, or step functions. They are also often monotonically increasing, continuous, differentiable and bounded.

2.1.1 Basic structure

For a given artificial neuron, let there be $m + 1$ inputs with signals x_0 through x_m and weights w_0 through w_m. Usually, the x_0 input is assigned the value +1, which makes it a bias input with $w_{k0} = b_k$. This leaves only m actual inputs to the neuron: from x_1 to x_m.

The output of the k_{th} neuron is:

$$y_k = \phi \left(\sum_{j=0}^{m} w_{kj} x_j \right) \tag{1}$$

Where ϕ (phi) is the transfer function.

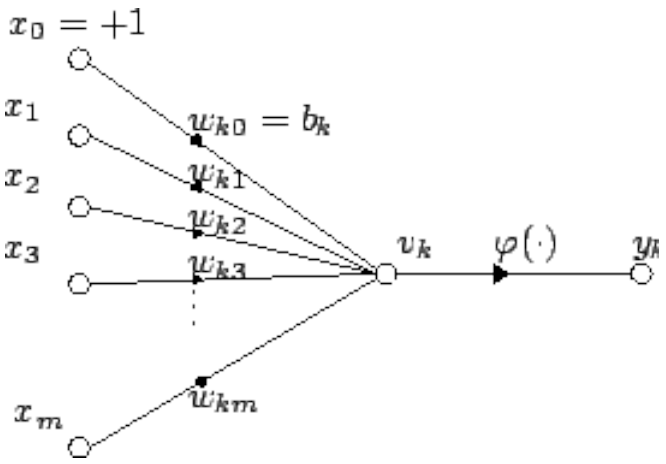


Figure 5: Representation of an artificial neuron.

The output is analogous to the axon of a biological neuron, and its value propagates to the input of the next layer, through a synapse. It may also exit the system, possibly as part of an output vector.

It has no learning process as such. Its transfer function weights are calculated and threshold value are predetermined.

2.1.2 Comparison to biological neurons

Artificial neurons are designed to mimic aspects of their biological counterparts.

1. **Dendrites-** In a biological neuron, the dendrites act as the input vector. These dendrites allow the cell to receive signals from a large (> 1000) number of neighboring neurons. As in the above mathematical treatment, each dendrite is able to perform "multiplication" by that dendrite's "weight value." The multiplication is accomplished by increasing or decreasing the ratio of synaptic neurotransmitters to signal chemicals introduced into the dendrite in response to the synaptic neurotransmitter. A negative multiplication effect can be achieved by transmitting signal inhibitors (i.e. oppositely charged ions) along the dendrite in response to the reception of synaptic neurotransmitters.

2. **Soma-** In a biological neuron, the soma acts as the summation function, seen in the above mathematical description. As positive and negative signals (exciting and inhibiting, respectively) arrive in the soma from the dendrites, the positive and negative ions are effectively added in summation, by simple virtue of being mixed together in the solution inside the cell's body.

3. **Axon-** The axon gets its signal from the summation behavior which occurs inside the soma. The opening to the axon essentially samples the electrical potential of the solution inside the soma. Once the soma reaches a certain potential, the axon will transmit an all-in signal pulse down its length. In this regard, the axon behaves as the ability for us to connect our artificial neuron to other artificial neurons.

2.1.3 Types of transfer functions

The transfer function of a neuron is chosen to have a number of properties which either enhance or simplify the network containing the neuron. Crucially, for instance, any multilayer perceptron using a linear transfer function has an equivalent single-layer network; a non-linear function is therefore necessary to gain the advantages of a multi-layer network.

Below, u refers in all cases to the weighted sum of all the inputs to the neuron, i.e. for n inputs,

$$u = \sum_{i=1}^{n} w_i x_i \tag{2}$$

where w is a vector of *synaptic weights* and $\mathbf{x}$ is a vector of inputs.

Step Function The output y of this transfer function is binary, depending on whether the input meets a specified threshold, θ. The "signal" is sent, i.e. the output is set to one, if the activation meets the threshold.

$$y = \begin{cases} 1 & \text{if } u \geq \theta \\ 0 & \text{if } u < \theta \end{cases}$$

Linear combination In this case, the output unit is simply the weighted sum of its inputs plus a bias term. A number of such linear neurons perform a linear transformation of the input vector. This is usually more useful in the first layers of a network. A number of analysis tools exist based on linear models, such as harmonic analysis, and they can all be used in neural networks with this linear neuron. The bias term allows us to make affine transformations to the data.

Sigmoid A fairly simple non-linear function, a sigmoid function such as the logistic function also has an easily calculated derivative, which can be important when calculating the weight updates in the network. It thus makes the network more easily manipulable mathematically, and was attractive to early computer scientists who needed to minimize the computational load of their simulations. **It is commonly seen in multilayer perceptrons using a backpropagation algorithm.**

2.2 Artificial Neural Network

In machine learning and cognitive science, artificial neural networks (ANNs) are a family of models inspired by biological neural networks (the central nervous systems of animals, in particular the brain) and are used to estimate or approximate functions that can depend on a large number of inputs and are generally unknown. Artificial neural networks are generally presented as systems of interconnected "neurons" which exchange messages between each other. The connections have numeric weights that can be tuned based on experience, making neural nets adaptive to inputs and capable of learning.

For example, a neural network for handwriting recognition is defined by a set of input neurons which may be activated by the pixels of an input image. After being weighted and transformed by a function (determined by the network's designer), the activations of these neurons are then passed on

to other neurons. This process is repeated until finally, an output neuron is activated. This determines which character was read.

Like other machine learning methods systems that learn from data neural networks have been used to solve a wide variety of tasks that are hard to solve using ordinary rule-based programming, including computer vision and speech recognition.

A basic model of neural network is displayed in figure 6.

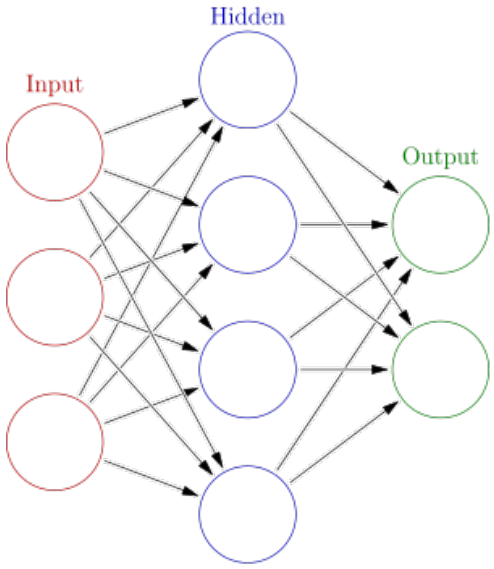


Figure 6: Basic model of artificial neural networks

2.3 Feedforward Neural Network

A feedforward neural network is an artificial neural network where connections between the units do not form a cycle. This is different from recurrent neural networks.

The feedforward neural network was the first and simplest type of artificial neural network devised. In this network, the information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) and to the output nodes. There are no cycles or loops in the network.

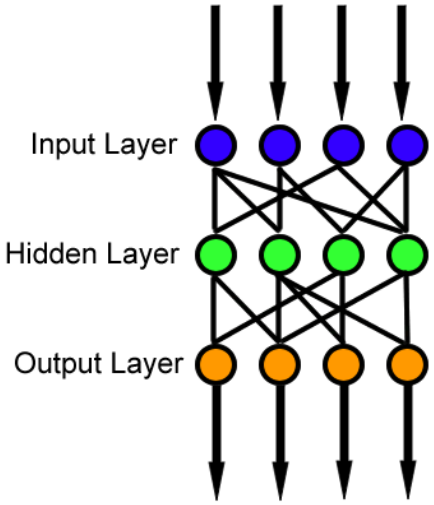


Figure 7: A simple Feedforward neural network.

2.3.1 Multi-layer perceptron

This class of networks consists of multiple layers of computational units, usually interconnected in a feed-forward way. Each neuron in one layer has directed connections to the neurons of the subsequent layer. In many applications the units of these networks apply a sigmoid function as an activation function.

The universal approximation theorem for neural networks states that every continuous function that maps intervals of real numbers to some output interval of real numbers can be approximated arbitrarily closely by a multi-layer perceptron with just one hidden layer. This result holds for a wide range of activation functions, e.g. for the sigmoidal functions.

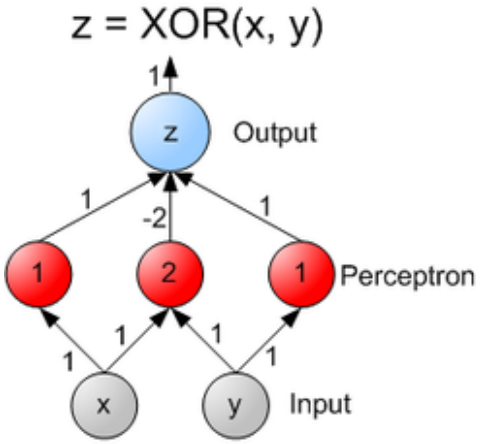


Figure 8: A multi-layer perceptron network learning XOR

2.3.2 Backpropagation

Backpropagation, an abbreviation for "backward propagation of errors", is a common method of training artificial neural networks used in conjunction with an optimization method such as gradient descent. The method calculates the gradient of a loss function with respect to all the weights in the network. The gradient is fed to the optimization method which in turn uses it to update the weights, in an attempt to minimize the loss function.

2.3.3 Gradient descent

Gradient descent is a first-order optimization algorithm. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or of the approximate gradient) of the function at the current point.

3 Importance of the project

Robotics technology is emerging at a rapid pace, offering new possibilities for automating tasks in many challenging applications, especially in autonomous self driving vehicles. A lot of parameters and conditions and other necessary things are needed to be considered to build a complete autonomous self driving vehicles, yet making the vehicle to learn to follow the line of the path is one of the basic building blocks to build a complete autonomous self driving vehicles. The main objective of the project is to train a network and apply it on a autonomous line following prototype which can navigate autonomously at any line consisting of any width.

Parameters of vehicles can be different from each other but the training techniques to follow a line on a surface will be the same for all of the test objects. So it is more convenient to train the network and set the weight than defining all the conditions to follow line.

4 Prototype construction

List and description of the items and electronic modules used in this project.

4.1 Arduino Nano

The Arduino Nano is a small, complete, and breadboard-friendly board based on the ATmega328 (Arduino Nano 3.x) or ATmega168 (Arduino Nano 2.x). It has more or less the same functionality of the Arduino Duemilanove, but in a different package. It lacks only a DC power jack, and works with a Mini-B USB cable instead of a standard one. The Nano was designed and is being produced by Gravitech.

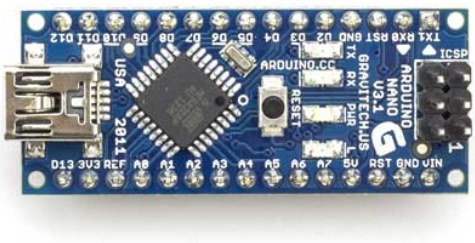


Figure 9: Arduino Nano Front View

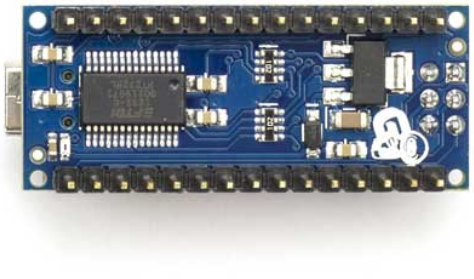


Figure 10: Arduino Nano Back View

4.1.1 Technical Specifications

Microcontroller	Atmel ATmega168 or ATmega328
Operating Voltage (logic level)	5 V
Input Voltage (recommended)	7-12 V
Input Voltage (limits)	6-20 V
Digital I/O Pins	14 (of which 6 provide PWM output)
Analog Input Pins	8
DC Current per I/O Pin	40 mA
Flash Memory	16 KB (ATmega168) or 32 KB (ATmega328) of which 2 KB used by bootloader
SRAM	1 KB (ATmega168) or 2 KB (ATmega328)
EEPROM	512 bytes (ATmega168) or 1 KB (ATmega328)
Clock Speed	16 MHz
Dimensions	0.73" x 1.70"
Length	45 mm
Width	18 mm
Weight	5 g

4.2 QTR-8A Reflectance Sensor Array

This sensor module has 8 IR LED/phototransistor pairs mounted on a 0.375" pitch. Pairs of LEDs are arranged in series to halve current consumption, and a MOSFET allows the LEDs to be turned off for additional sensing or power-savings options. Each sensor provides a separate analog voltage output.

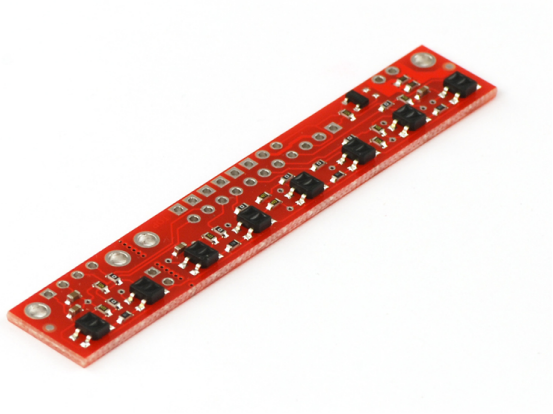


Figure 11: Pololu QTR-8A Reflectance Sensor Array

4.3 L298N Motor Driver

To drive two motors, L298N stepper motor driver breakout was used.

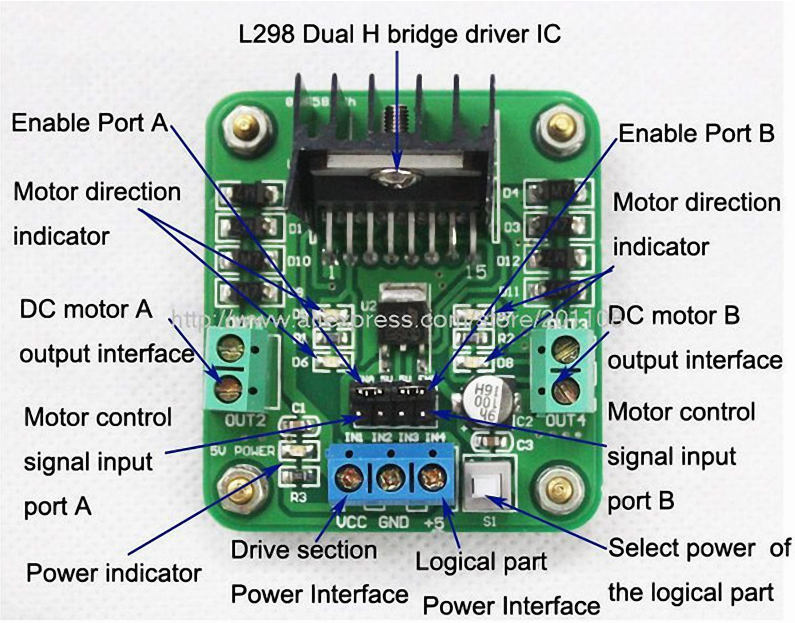


Figure 12: L298N DC Motor Driver

4.4 Lithium-Polymer Battery

A 3 cell Li-Po battery was used with 800mAh capacity.

Figure 13: 3 Cell 800mAh Li-Po Battery

4.5 Bluetooth Module

HC-05 module is an easy to use Bluetooth SPP (Serial Port Protocol) module, designed for transparent wireless serial connection setup. Serial port Bluetooth module is fully qualified Bluetooth V2.0+EDR (Enhanced Data Rate) 3Mbps Modulation with complete 2.4GHz radio transceiver and baseband. It uses CSR Bluecore 04-External single chip Bluetooth

system with CMOS technology and with AFH(Adaptive Frequency Hopping Feature). It has the footprint as small as $12.7mm \times 27mm$.

Hardware Features

1. Typical $-80dBm$ sensitivity
2. Up to $+4dBm$ RF transmit power
3. Low Power 1.8V Operation
4. UART interface with programmable baud rate
5. With edge connector

Figure 14: HC-05 Bluetooth Module

4.6 Breadboard

For completing the circuit of the prototype I used small breadboard to connect the wires.

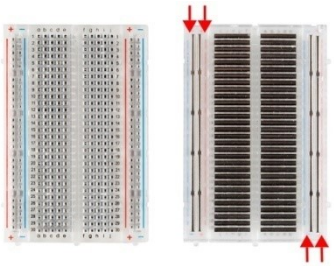


Figure 15: Half size breadboard (clear)

4.7 Premium Jumper Wire

Male to male and male to female jumper connectors to connect the components together.

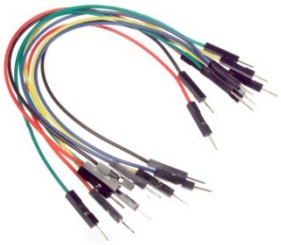


Figure 16: Connecting wires

5 Sensor positioning of the prototype

5.0.1 Bottom view

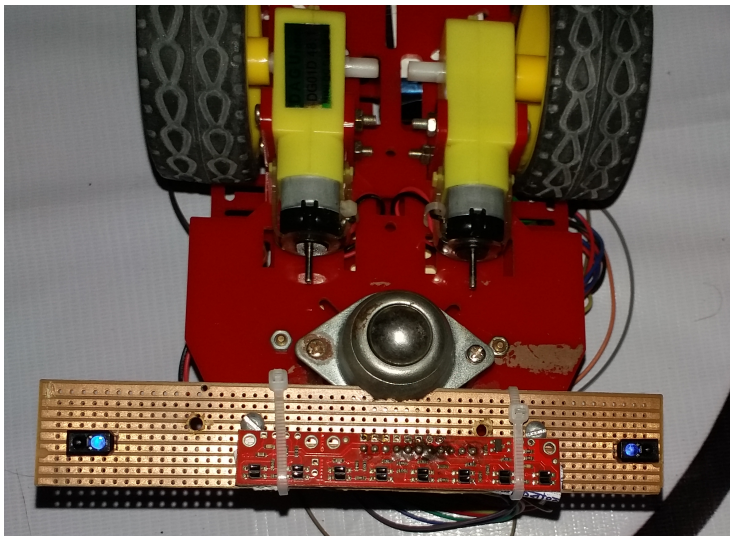


Figure 17: Placement of IR sensors

5.0.2 Top view

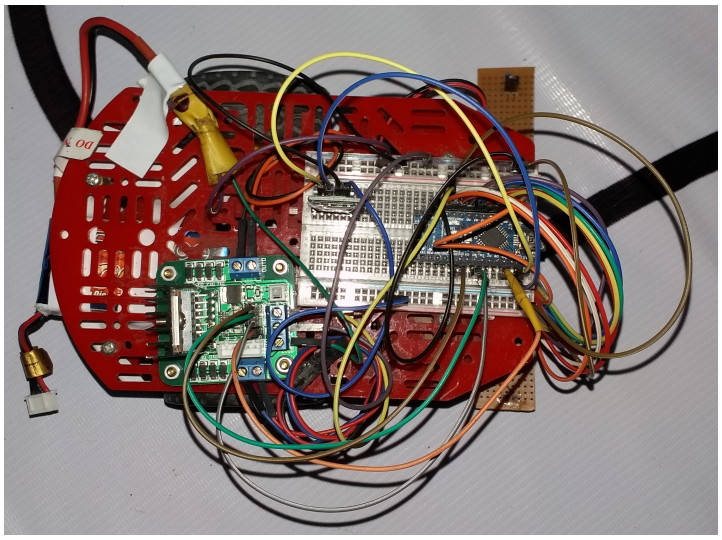
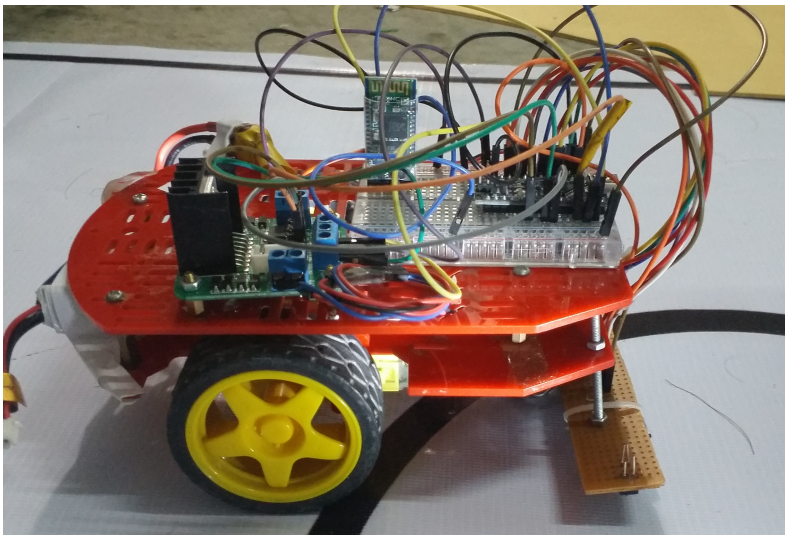


Figure 18: Bottom view of the robot.

5.0.3 Side view



6 Circuit diagram

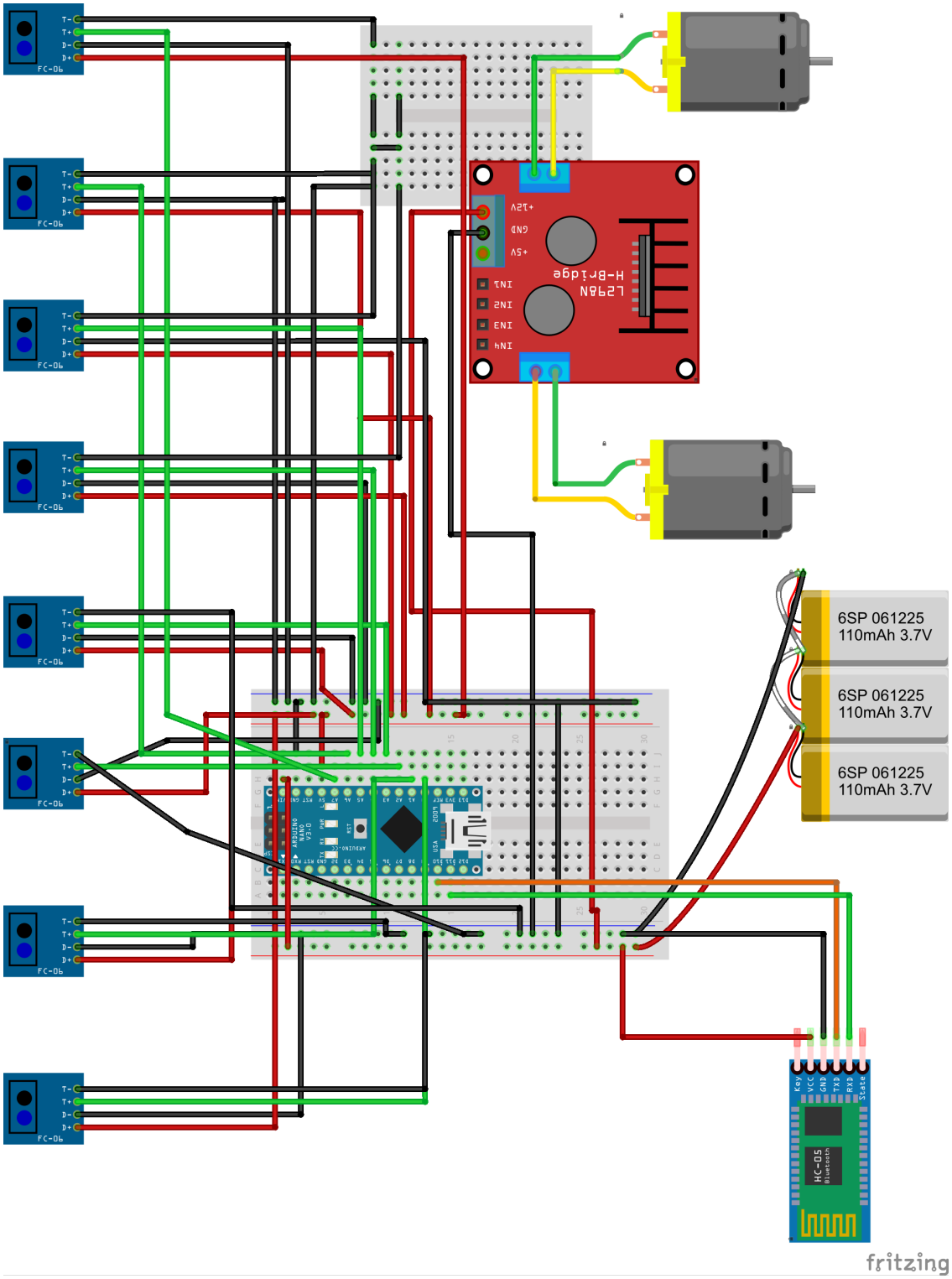


Figure 19: Circuit diagram of the robotic platform.

7 Data collection process

7.1 Processing code for driving and collecting data wirelessly

```
import processing.serial.*;
int value = 0;
int width = 50;
int height = 50;
final int baud = 9600;

Serial arduino;

void setup(){
  String arduinoPort = "COM12";
  println(Serial.list());
  arduino = new Serial(this, arduinoPort, baud);
}

char old_key = '0';
char new_key = '1';

//Checking if any key was pressed, initializing with a 'false'
boolean wasPressed = false;

void serialEvent(Serial port){
  String inByte = port.readStringUntil('\n');
  if (inByte != null) println(inByte);
}

void draw(){
  if (keyPressed == true){
    new_key = key;
    if (new_key != old_key) {
      arduino.write(key);
    }
    wasPressed = true;
  } else {
    if (wasPressed == true){
```

```
      new_key = 'z';
      arduino.write('z');
      wasPressed = false;
    }
  }
  old_key = new_key;
}
```

7.2 Arduino code for the robotic prototype

7.2.1 LineFollower.h

```
#ifndef LINE_FOLLOWER_H_
#define LINE_FOLLOWER_H_
#include <Arduino.h>
#include <StandardCplusplus.h>
#include <vector>
#include <serstream>
#include <iterator>
#include <SoftwareSerial.h>

using namespace std;

//If inverse logic is defined
#if defined(INVERSE_LOGIC)
#define INSIDE_LINE 0
#define OUTSIDE_LINE 1
#else
#define INSIDE_LINE 1
#define OUTSIDE_LINE 0
#endif

///define front and back pin
#define FRONT_PIN 0
#define BACK_PIN 1

#define BAUD 9600
#define DEFAULT_SPEED 150
#define MULTIPLIER 1000
```

```
#define RX 10
#define TX 11

//Delay macro
#define SHORT_DELAY 100
#define LONG_DELAY 500

//Threshold
#define THRESHOLD 500

//add and subtract from or to this speed
#define spd 85
#define speed_factor 10
#define spd_factor 4
#define add_speed 60

//define input and output macro
#define input(pin) pinMode(pin, INPUT)
#define output(pin) pinMode(pin, OUTPUT)

//Define high low and pwm macro
#define pwm(pin, value) analogWrite(pin, value)
#define on(pin) digitalWrite(pin, HIGH)
#define off(pin) digitalWrite(pin, LOW)

#define short_delay delay(SHORT_DELAY)
#define long_delay delay(LONG_DELAY)

typedef unsigned int u_int;

class LineFollower
{
private:

  u_int leftMotorPin[2];
  u_int rightMotorPin[2];
  u_int defaultSpeed;
  vector <u_int> sensors;
  u_int leftSpeed;
```

```
  u_int rightSpeed;
  char command;
  u_int activeSensors;
public:
//Digital and analog reading array
vector <u_int> digitalReading;
vector <u_int> analogReading;

    //Constructors
    LineFollower(void) :
defaultSpeed(DEFAULT_SPEED),
activeSensors(0)
{
leftSpeed = DEFAULT_SPEED;
rightSpeed = DEFAULT_SPEED;
};

    LineFollower(u_int *lm, u_int *rm, vector <u_int> &s);

    //Initializer
    void serial_init(u_int b);
    void init(void);
void sensors_init(void);
void sensors_init(vector <u_int> &s);
void set_sensors_pins(vector <u_int> s);

    //Motor config functions
    void set_motors_pins(u_int *lm, u_int *rm);
    void set_motors(int lspeed, int rspeed);
    void motors_init(void);
void stop(void);
void set_speed(u_int dSpeed);

//Extra motor controls
void forward(u_int lspeed, u_int rspeed);
void backward(u_int lspeed, u_int rspeed);
void left(u_int rspeed);
void right(u_int lspeed);
void anticlockwise(u_int lspeed, u_int rspeed);
void clockwise(u_int lspeed, u_int rspeed);
```

```
//Bluetooth control
void wireless_control(void);
void wireless_control(bool debug_mode);

//Debugging sensors
void debug(void);
void wireless_debug(void);

//Line Following control
void read_sensors(void);
int read_line(void);
void clear_reading(void);
void conditional_drive(void);
};

extern LineFollower LineFollowingRobot;
extern SoftwareSerial Bluetooth;
#endif
```

7.2.2 LineFollower.cpp

```
#include <LineFollower.h>

//Objects created for line following robot and bluetooth
LineFollower LineFollowingRobot;
SoftwareSerial Bluetooth(RX, TX);

namespace std
{
  ohserialstream cout(Serial);
}

using namespace std;

//Initialize everything
void LineFollower::init(void)
{
  motors_init();
  sensors_init();
```

```
  serial_init(BAUD);
  defaultSpeed = DEFAULT_SPEED;
  input(0);
  input(7);
}

//========== MOTOR SECTION ==============
//Initialize motors
void LineFollower::motors_init(void)
{
  for (u_int i = 0; i < 2; i++)
  {
    pinMode(leftMotorPin[i], OUTPUT);
    pinMode(rightMotorPin[i], OUTPUT);
  }
}

//Run the motors
void LineFollower::set_motors(int lspeed, int rspeed)
{
  //Forward || Left || Right
  if (lspeed >= 0 && rspeed >= 0)
  {
    leftSpeed = lspeed;
    rightSpeed = rspeed;
    pwm(leftMotorPin[FRONT_PIN], leftSpeed);
    pwm(rightMotorPin[FRONT_PIN], rightSpeed);
    off(leftMotorPin[BACK_PIN]);
    off(rightMotorPin[BACK_PIN]);
  }
  //Right || Clockwise
  else if (lspeed >= 0 && rspeed < 0)
  {
    leftSpeed = lspeed;
    rightSpeed = -rspeed;
    pwm(leftMotorPin[FRONT_PIN], leftSpeed);
    off(leftMotorPin[BACK_PIN]);
    pwm(rightMotorPin[BACK_PIN], rightSpeed);
    off(rightMotorPin[FRONT_PIN]);
  }
```

```
  //Left || Anticlockwise
  else if (lspeed < 0 && rspeed >= 0)
  {
    leftSpeed = -lspeed;
    rightSpeed = rspeed;
    pwm(rightMotorPin[FRONT_PIN], rightSpeed);
    off(rightMotorPin[BACK_PIN]);
    pwm(leftMotorPin[BACK_PIN], leftSpeed);
    off(leftMotorPin[FRONT_PIN]);
  }
  //Backward
  else if (lspeed < 0 && rspeed < 0)
  {
    leftSpeed = -lspeed;
    rightSpeed = -rspeed;
    pwm(leftMotorPin[BACK_PIN], leftSpeed);
    pwm(rightMotorPin[BACK_PIN], rightSpeed);
    off(leftMotorPin[FRONT_PIN]);
    off(rightMotorPin[FRONT_PIN]);
  }
  //Stop
  else stop();
}

//Extra motor controls
void LineFollower::forward(u_int lspeed, u_int rspeed)
{
  leftSpeed = lspeed;
  rightSpeed = rspeed;
  pwm(leftMotorPin[FRONT_PIN], leftSpeed);
  pwm(rightMotorPin[FRONT_PIN], rightSpeed);
  off(leftMotorPin[BACK_PIN]);
  off(rightMotorPin[BACK_PIN]);
}

void LineFollower::backward(u_int lspeed, u_int rspeed)
{
  leftSpeed = lspeed;
  rightSpeed = rspeed;
  pwm(leftMotorPin[BACK_PIN], leftSpeed);
```

```
  pwm(rightMotorPin[BACK_PIN], rightSpeed);
  off(leftMotorPin[FRONT_PIN]);
  off(rightMotorPin[FRONT_PIN]);
}

void LineFollower::left(u_int rspeed)
{
  rightSpeed = rspeed;
  off(leftMotorPin[BACK_PIN]);
  off(leftMotorPin[FRONT_PIN]);
  pwm(rightMotorPin[FRONT_PIN], rightSpeed);
  off(rightMotorPin[BACK_PIN]);
}

void LineFollower::right(u_int lspeed)
{
  leftSpeed = lspeed;
  off(rightMotorPin[FRONT_PIN]);
  off(rightMotorPin[BACK_PIN]);
  pwm(leftMotorPin[FRONT_PIN], leftSpeed);
  off(leftMotorPin[BACK_PIN]);
}

void LineFollower::clockwise(u_int lspeed, u_int rspeed)
{
  leftSpeed = lspeed;
  rightSpeed = rspeed;
  pwm(leftMotorPin[FRONT_PIN], leftSpeed);
  off(leftMotorPin[BACK_PIN]);
  pwm(rightMotorPin[BACK_PIN], rightSpeed);
  off(rightMotorPin[FRONT_PIN]);
}

void LineFollower::anticlockwise(u_int lspeed, u_int rspeed)
{
  leftSpeed = lspeed;
  rightSpeed = rspeed;
  pwm(leftMotorPin[FRONT_PIN], leftSpeed);
  off(leftMotorPin[BACK_PIN]);
  pwm(rightMotorPin[BACK_PIN], rightSpeed);
```

```
  off(rightMotorPin[FRONT_PIN]);
}

//Set the motor pins
void LineFollower::set_motors_pins(u_int *lm, u_int *rm)
{
  for (u_int i = 0; i < 2; i++){
    leftMotorPin[i] = lm[i];
    rightMotorPin[i] = rm[i];
  }
}

void LineFollower::stop(void)
{
  for (u_int i = 0; i < 2; i++){
    off(leftMotorPin[i]);
    off(rightMotorPin[i]);
  }
}

void LineFollower::set_speed(u_int dSpeed)
{
  defaultSpeed = dSpeed;
}

/********* SENSOR **************/
//Initialize sensors
void LineFollower::sensors_init(void)
{
  for (u_int i = 0; i < sensors.size(); i++)
    pinMode(sensors[i], INPUT);
}

void LineFollower::sensors_init(vector <u_int> &s)
{
  sensors = s;
  for (u_int i = 0; i < sensors.size(); i++) pinMode(sensors[i],
INPUT);
}
```

```
void LineFollower::set_sensors_pins(vector <u_int> s)
{
  sensors_init(s);
}

/*== DEBUG SECTION*/
//Print sensor value
void LineFollower::debug(void)
{
  long_delay;
  cout << " ======= ANALOG READING ===== " << endl;
  for (u_int i = 0; i < sensors.size(); i++)
  {
    cout << "ir [" << (i + 1) << "] = " << analogRead(sensors[i])
<< endl;
    short_delay;
  }
  cout << " ====== ANALOG READING END ======" << endl;
  cout << endl << endl;
  short_delay;
  cout << " ======= DIGITAL READING ======= " << endl;
  read_sensors();
  short_delay;
  for (u_int i = 0; i < sensors.size(); i++)
    cout << digitalReading[i] << " ";
  cout << endl;
  cout << " ======= DIGITAL READING END ==== " << endl;
  cout << endl << endl;
  long_delay;
}

void LineFollower::wireless_debug(void)
{
  long_delay;
  Bluetooth.println("======= ANALOG READING ===== ");
  for (u_int i = 0; i < sensors.size(); i++){
  Bluetooth.println("ir [" + String(i + 1) + "] = " +
   String(analogRead(sensors[i])));
    short_delay;
  }
```

```
  Bluetooth.println("======= ANALOG READING END ===== ");
  Bluetooth.println();
  Bluetooth.println();
  short_delay;

  Bluetooth.println(" ==== DIGITAL READING START === ");
  read_sensors();
  short_delay;

  for (u_int i = 0; i < sensors.size(); i++)
  Bluetooth.print(digitalReading[i] + String(" "));

  Bluetooth.println();
  Bluetooth.println("====== DIGITAL READING END ======");
  Bluetooth.println();
  Bluetooth.println("Calculated position: " + String(read_line()));
  Bluetooth.println();
  long_delay;
}

// LINE FOLLOWING CONTROL
void LineFollower::read_sensors(void)
{
  digitalReading.clear();
  analogReading.clear();
  activeSensors = 0;
  for (u_int i = 0; i < sensors.size(); i++) {
    u_int reading = analogRead(sensors[i]);
    analogReading.push_back(reading);
    if (reading > THRESHOLD){
      digitalReading.push_back(INSIDE_LINE);
      activeSensors++;
    }
    else
      digitalReading.push_back(OUTSIDE_LINE);
  }
}

void LineFollower::clear_reading(void)
{
```

```
  digitalReading.clear();
  analogReading.clear();
}

int LineFollower::read_line(void)
{
  u_int totalWeight = 0;
  read_sensors();
  for (u_int i = 0; i < sensors.size(); i++)
  {
    totalWeight += (i + 1) * digitalReading[i] * MULTIPLIER;
  }
  return (totalWeight / activeSensors);
}

//Driving definition
void LineFollower::conditional_drive(void)
{
  read_line();
  if (digitalReading[3] == 1 || digitalReading[4] == 1)
    forward(defaultSpeed, defaultSpeed);
  else if (digitalReading[2] == 1)
    forward(0, add_speed + speed_factor * 1.5);
  else if (digitalReading[1] == 1)
    forward(0, add_speed + speed_factor * 2.5);
  else if (digitalReading[0] == 1)
    forward(0, add_speed + speed_factor * 3.5);
  else if (digitalReading[5] == 1)
    forward(add_speed + speed_factor * 1.5, 0);
  else if (digitalReading[6] == 1)
    forward(add_speed + speed_factor * 2.5, 0);
  else if (digitalReading[7] == 1)
    forward(add_speed + speed_factor * 3.5, 0);
  else if (digitalReading[0] == 1 && digitalReading[1] == 1)
    forward(0, add_speed + speed_factor * 3);
  else if (digitalReading[6] == 1 && digitalReading[7] == 1)
    forward(add_speed + speed_factor * 3, 0);
  else stop();
}
```

```
/*
//  ========== CONSTRUCTOR ==========
*/
//Ctor
LineFollower::LineFollower(u_int *lm, u_int *rm, vector <u_int>
&s)
: defaultSpeed(DEFAULT_SPEED),
  activeSensors(0) {
  sensors = s;
  for (u_int i = 0; i < 2; i++)
  {
    leftMotorPin[i] = lm[i];
    rightMotorPin[i] = rm[i];
  }
}

//Initialize serial communication
void LineFollower::serial_init(u_int baud)
{
  Serial.begin(baud);
  Bluetooth.begin(baud);
}

//========== WIRELESS CONTROL ========
void LineFollower::wireless_control(void)
{
  if (Bluetooth.available() > 0)
   command = Bluetooth.read();
  switch (command)
  {
    case 'w':
      forward(defaultSpeed, defaultSpeed);
      break;
    case 's':
      backward(defaultSpeed, defaultSpeed);
      break;
    case 'a':
      left(defaultSpeed);
      break;
    case 'd':
```

```
      right(defaultSpeed);
      break;
    case 'r':
      wireless_debug();
      break;
    default:
      stop();
      break;
  }
}

void LineFollower::wireless_control(bool debug_mode)
{
  if (!debug_mode) wireless_control();
  else
  {
    if (Bluetooth.available() > 0){
      command = Bluetooth.read();
      Serial.println("I received: " + String(command));
    }
    switch (command){
      Bluetooth.println(command);
      case 'w':
        forward(defaultSpeed, defaultSpeed);
        break;
      case 's':
        backward(defaultSpeed, defaultSpeed);
        break;
      case 'a':
        left(defaultSpeed);
        break;
      case 'd':
        right(defaultSpeed);
        break;
      case 'r':
        wireless_debug();
        break;
      default:
        stop();
        break;
```

```
    }
  }
}
```

7.2.3 ANNLineFollower.ino

```
#include <LineFollower.h>

//Sensor pins
vector <u_int> qtr{0, 1, 2, 3, 4, 5, 6, 7};

u_int lm[] = {5, 3};
u_int rm[] = {6, 9};

LineFollower lineFollower(lm, rm, qtr);

void setup()
{
  LineFollowingRobot.set_motors_pins(lm, rm);
  LineFollowingRobot.set_sensors_pins(qtr);
  LineFollowingRobot.init();
  LineFollowingRobot.set_speed(75);
}

void loop() {
 LineFollowingRobot.wireless_control(true);
}
```

7.2.4 Data collection using processing

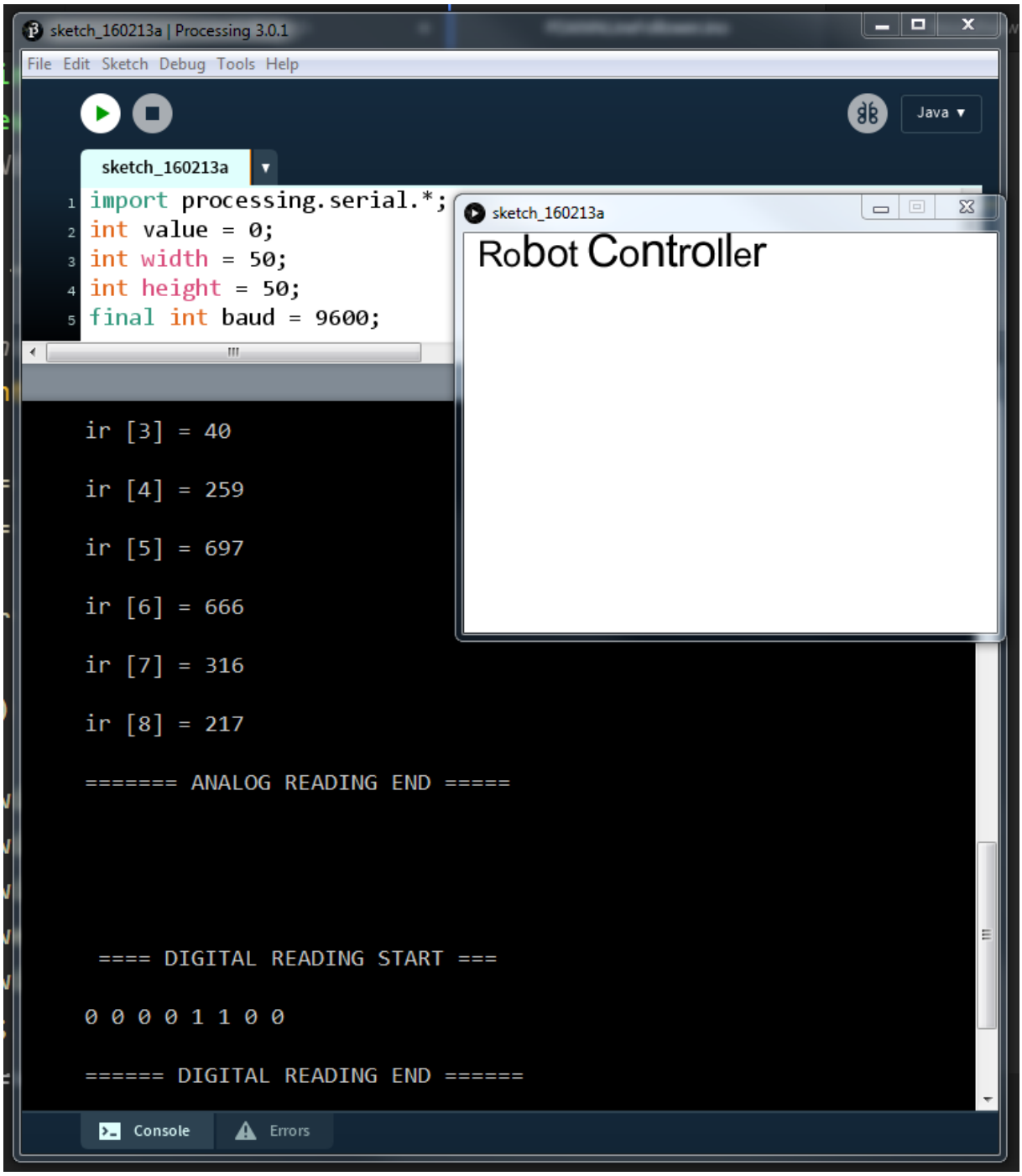
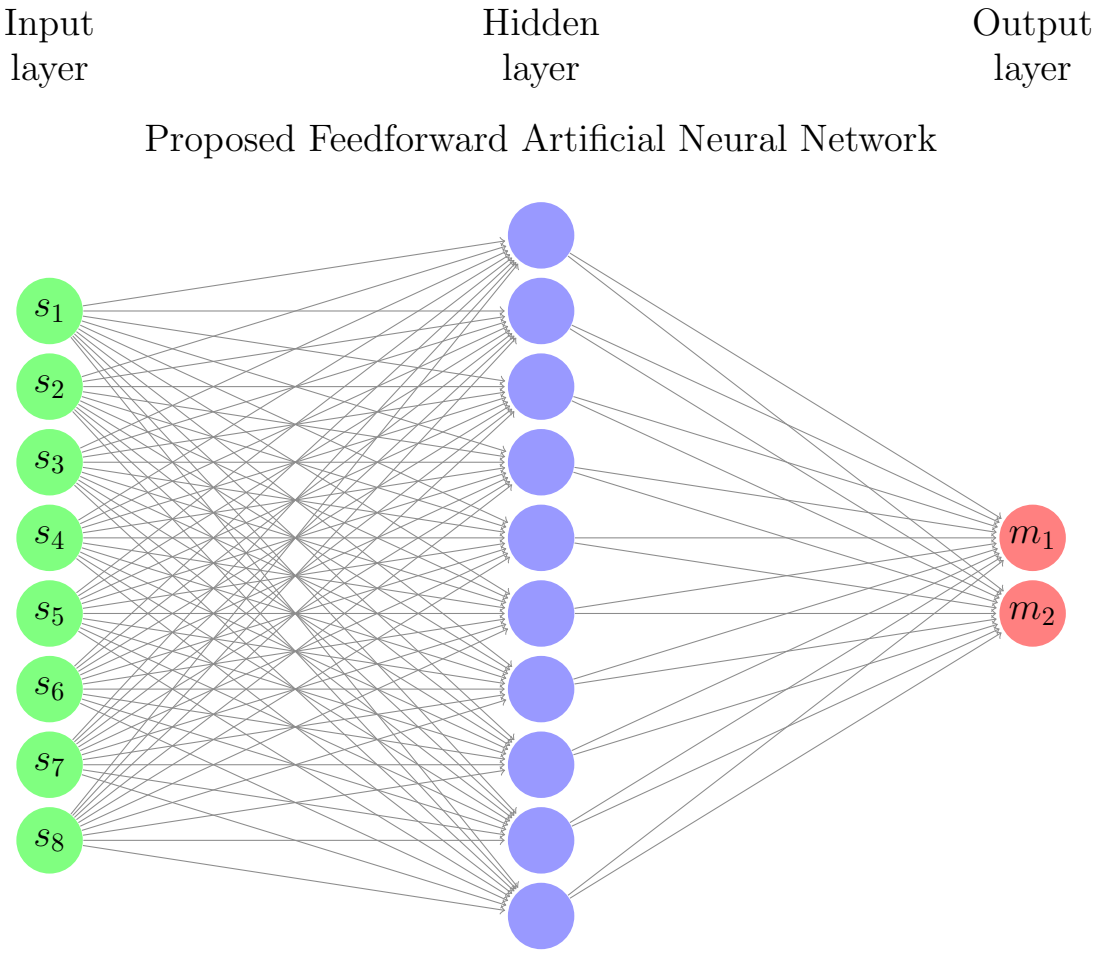


Figure 20: Robot controller & data collector using processing

8 Proposed network configuration & training

8.1 Diagram of the proposed network



8.2 Collected data

Table 1: Collected input and target data

ir1	ir2	ir3	ir4	ir5	ir6	ir7	ir8	left target	right target
0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	75	75
0	0	0	0	1	0	0	0	75	75
0	0	0	1	1	0	0	0	75	75
0	0	1	1	1	0	0	0	75	75
0	0	0	1	1	1	0	0	75	75
0	0	1	0	0	0	0	0	0	81
0	0	1	1	0	0	0	0	0	81
0	1	1	1	0	0	0	0	0	81
0	1	0	0	0	0	0	0	0	85
1	1	0	0	0	0	0	0	0	85
1	1	1	0	0	0	0	0	0	85
1	0	0	0	0	0	0	0	0	89
0	0	0	0	0	1	0	0	81	0
0	0	0	0	1	1	0	0	81	0
0	0	0	0	1	1	1	0	81	0
0	0	0	0	0	0	1	0	85	0
0	0	0	0	0	0	1	1	85	0
0	0	0	0	0	1	1	1	85	0
0	0	0	0	0	0	0	1	89	0

8.3 Training in MATLAB

8.3.1 Code used to train the network in MATLAB

```
net = newff( minmax(inputs) , [10 2], {‘tansig’ ‘tansig’},‘trainlm’,
‘learngdm’,‘mse’);
[trainInd ,valInd, testInd] = dividerand(0.70 ,0.15 ,0.15);
net = train(net ,inputs ,targets);
y = net(inputs);
perf = perform(net, y, targets);
e = targets - y;
view(net);
ploterrhist(e);
```

The training was being stopped by observing the mean square error

value, gradient value and regression curve. The plots of these parameters are given in figure 21 and 22.

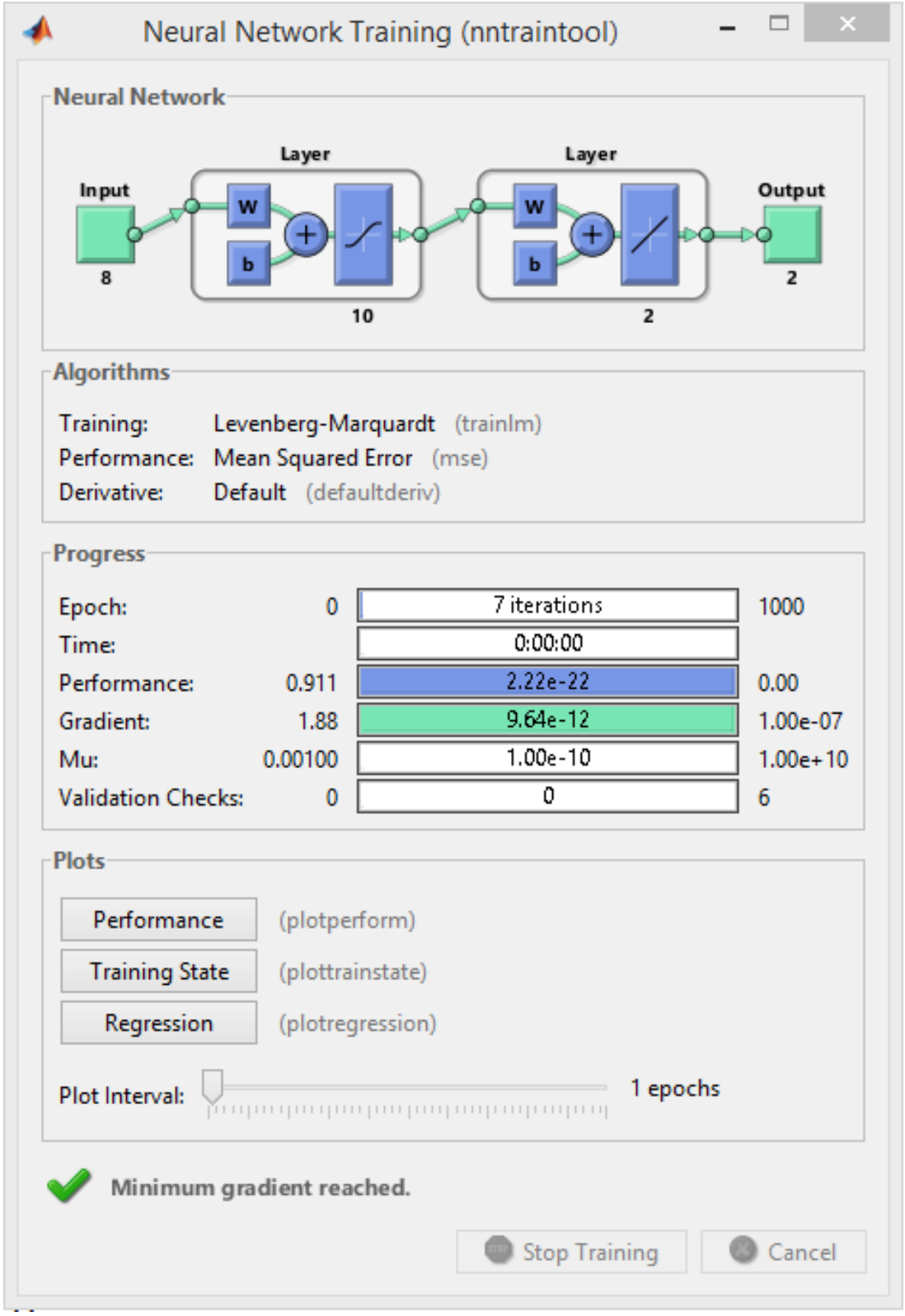


Figure 21: Training of the network

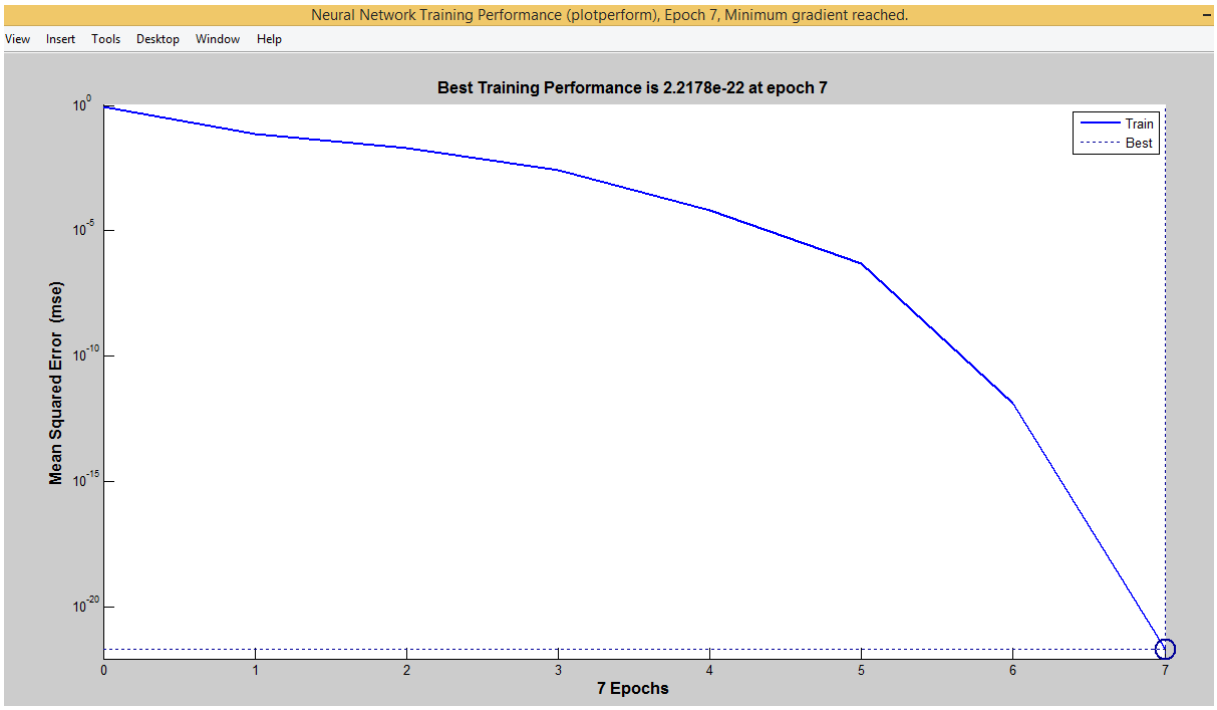


Figure 22: Performance of the trained network

8.4 Extraction of weights and implementation on arduino

After successful training the weights were extracted using the following program in MATLAB.

```
hidden = net.LW
input = net.IW
```

Neural network with extracted weights in *NeuralNetworkConfig.h* file.

8.4.1 NeuralNetworkConfig.h

```
#ifndef NEURAL_NETWORK_CONFIG_H_
#define NEURAL_NETWORK_CONFIG_H_
#include <LineFollower.h>

const int PatternCount = 1;
const int InputNodes = 8;
const int HiddenNodes = 10;
const int OutputNodes = 2;
```

```
double Input[PatternCount][InputNodes];

double Hidden[HiddenNodes];
double Output[OutputNodes];
double HiddenWeights[InputNodes][HiddenNodes];
double OutputWeights[HiddenNodes][OutputNodes];

//Variables
double Accum;

void updateInput(void)
{
  LineFollowingRobot.read_sensors();
  for (u_int i = 0; i < LineFollowingRobot.digitalReading.size();
i++){
    Input[0][i] = LineFollowingRobot.digitalReading[i];
  }
}

void weights(void)
{
  //Input to Hidden Weights
  //Hidden Weights for first IR
  HiddenWeights[0][0] =0.025113 ;
  HiddenWeights[0][1] =-0.761469 ;
  HiddenWeights[0][2] = -0.054762 ;
  HiddenWeights[0][3] = 1.929800;
  HiddenWeights[0][4] =  0.896791;
  HiddenWeights[0][5] = -0.642489 ;
  HiddenWeights[0][6] = 2.283807 ;
  HiddenWeights[0][7] = -3.604985 ;
  HiddenWeights[0][8] = 1.723457 ;
  HiddenWeights[0][9] =  -1.394811;

  //Second
  HiddenWeights[1][0] = -0.321062 ;
  HiddenWeights[1][1] = -0.724505 ;
  HiddenWeights[1][2] =  1.516371;
  HiddenWeights[1][3] = -0.767078 ;
  HiddenWeights[1][4] = -1.187555 ;
```

```
HiddenWeights[1][5] =  1.021865;
HiddenWeights[1][6] =  -1.131159;
HiddenWeights[1][7] = -1.998590 ;
HiddenWeights[1][8] =  2.275912;
HiddenWeights[1][9] =  0.562050;

//Third
HiddenWeights[2][0] =  -0.786959;
HiddenWeights[2][1] =  -0.081098;
HiddenWeights[2][2] = -1.879575 ;
HiddenWeights[2][3] = 0.454331 ;
HiddenWeights[2][4] =  -1.062709;
HiddenWeights[2][5] = -1.092372 ;
HiddenWeights[2][6] =  -0.779593;
HiddenWeights[2][7] = -1.295421 ;
HiddenWeights[2][8] =  -1.352029;
HiddenWeights[2][9] =  1.942178;

//Fourth
HiddenWeights[3][0] = -1.649061 ;
HiddenWeights[3][1] = 2.324182 ;
HiddenWeights[3][2] = 0.847181 ;
HiddenWeights[3][3] =  -2.225811;
HiddenWeights[3][4] = 0.355119 ;
HiddenWeights[3][5] = 0.863736 ;
HiddenWeights[3][6] = 1.617657 ;
HiddenWeights[3][7] = -0.254025 ;
HiddenWeights[3][8] = 0.677678 ;
HiddenWeights[3][9] = 1.868622 ;

//Fifth
HiddenWeights[4][0] = -1.018758 ;
HiddenWeights[4][1] = -0.144537 ;
HiddenWeights[4][2] = -2.582041 ;
HiddenWeights[4][3] =  -1.409654;
HiddenWeights[4][4] = -0.138546 ;
HiddenWeights[4][5] = 1.797844;
HiddenWeights[4][6] = 1.701960 ;
HiddenWeights[4][7] = 0.621565 ;
HiddenWeights[4][8] = -1.213992 ;
```

```
HiddenWeights[4][9] = 1.645741 ;

//Sixth
HiddenWeights[5][0] =  0.687452;
HiddenWeights[5][1] =  -1.621133;
HiddenWeights[5][2] =  1.418583;
HiddenWeights[5][3] = 1.517330 ;
HiddenWeights[5][4] =  1.647353;
HiddenWeights[5][5] = 0.370115 ;
HiddenWeights[5][6] = -0.048096 ;
HiddenWeights[5][7] =  0.469378;
HiddenWeights[5][8] = 0.525366 ;
HiddenWeights[5][9] = -0.811420 ;

//Seventh
HiddenWeights[6][0] = 0.153970 ;
HiddenWeights[6][1] = -1.806422 ;
HiddenWeights[6][2] =  0.755054;
HiddenWeights[6][3] =  -1.545513;
HiddenWeights[6][4] = -1.765307 ;
HiddenWeights[6][5] = 0.110509 ;
HiddenWeights[6][6] = -0.177388 ;
HiddenWeights[6][7] =  1.461557;
HiddenWeights[6][8] =  0.563469;
HiddenWeights[6][9] = 0.697234 ;

//Eighth
HiddenWeights[7][0] =  -1.936623;
HiddenWeights[7][1] =  -2.409954;
HiddenWeights[7][2] =  0.318658;
HiddenWeights[7][3] = 0.113455 ;
HiddenWeights[7][4] = 2.136323 ;
HiddenWeights[7][5] =  1.984139;
HiddenWeights[7][6] =  0.013263;
HiddenWeights[7][7] =  1.487560;
HiddenWeights[7][8] =  1.346021;
HiddenWeights[7][9] = 0.425384 ;

//Hidden to Output weights
//1st
```

```
  OutputWeights[0][0] =-1.252983 ;
  OutputWeights[0][1] = 0.715766;

  //2nd
  OutputWeights[1][0] = -0.421231;
  OutputWeights[1][1] =0.058311 ;

  //3rd
  OutputWeights[2][0] =-0.205985 ;
  OutputWeights[2][1] =-0.834553 ;

  //4th
  OutputWeights[3][0] = -0.814466 ;
  OutputWeights[3][1] = -0.548837;

  //5th
  OutputWeights[4][0] =0.521854 ;
  OutputWeights[4][1] = 0.250546;

  //6th
  OutputWeights[5][0] = -2.152792 ;
  OutputWeights[5][1] = 0.659315;

  //7th
  OutputWeights[6][0] = -0.306374;
  OutputWeights[6][1] =  0.252961;

  //8th
  OutputWeights[7][0] =0.268343 ;
  OutputWeights[7][1] = -0.854261 ;

  //9th
  OutputWeights[8][0] =  -2.098765;
  OutputWeights[8][1] = 1.391521;

  //10th
  OutputWeights[9][0] = -0.595847;
  OutputWeights[9][1] = -0.996570;
}
```

```
void calculateOutput(void)
{
  for(int p = 0 ; p < PatternCount ; p++ )
  {
    for(int i = 0 ; i < HiddenNodes ; i++ ) {
      Accum = HiddenWeights[InputNodes][i];
      for(int j = 0 ; j < InputNodes ; j++ ) {
        Accum += Input[p][j] * HiddenWeights[j][i] ;
      }
      // Hidden[i] = 1.0/(1.0 + exp(-Accum)) ;

      //Tansig
      Hidden[i] = (2.0/ (1 + exp(-2 * Accum))) - 1;
    }

    for(int i = 0 ; i < OutputNodes ; i++ ) {
      Accum = OutputWeights[HiddenNodes][i] ;
      for(int j = 0 ; j < HiddenNodes ; j++ ) {
        Accum += Hidden[j] * OutputWeights[j][i] ;
      }
      //Tansig
     Output[i] = (2.0/ (1 + exp(-2 * Accum))) - 1;
    }
  }
}

void neural_network_drive(void)
{
  updateInput();
  calculateOutput();
  int lspeed = Output[0] * 100;
  int rspeed = Output[1] * 100;

  //Reduced speed
  lspeed -= 15;
  rspeed -= 15;
  if (lspeed < 0) lspeed = -lspeed;
  if (rspeed < 0) rspeed = -rspeed;
  LineFollowingRobot.forward(lspeed, rspeed);
}
```

```
#endif
```

9 Testing the network

Following arduino program was used to test the network, it is the modified version of the wireless control program.

9.1 ANNLineFollower.ino

```
#include <LineFollower.h>
#include <NeuralNetworkConfig.h>

//Sensor pins
vector <u_int> qtr{0, 1, 2, 3, 4, 5, 6, 7};

u_int lm[] = {5, 3};
u_int rm[] = {6, 9};

LineFollower lineFollower(lm, rm, qtr);

void setup()
{
  LineFollowingRobot.set_motors_pins(lm, rm);
  LineFollowingRobot.set_sensors_pins(qtr);
  LineFollowingRobot.init();
  lineFollower.init();
  weights();
}

void loop() { neural_network_drive(); }
```

10 Discussion

Speed of the robot was reduced due to the huge number of calculations were required to move each step. If the speed of the robot was high, it would not follow the line properly, this problem can be tackled using less number of nodes in hidden layer, but it can also reduce the performance of

the current network. So the speed was reduced by keeping higher number of nodes in the hidden layer. From figure 21 it can be seen that the error value was minimized to minimum within 7 iterations. This artificial neural network based robot can follow any width of line where differential drive based or proportional derivative controller based robots are needed to be tuned after changing the track.

In this project, the trained network was implemented successfully and desired output was observed.

11 Future Work

In future, this network can be trained with more inputs and outputs, at some situations, following line can be hard due to the complexity of the track. The track may have discontinued line or other angles e.g acute, right etc. To follow this kind of line, the robotic platform may need to calculate the delay time automatically to rotate to fix the position. Then the output layer will have three nodes, two for left and right motor output and another one will be for delay time.

12 References

1. Artificial Neural Network, Wikipedia https://en.wikipedia.org/wiki/Artificial_neural_network

2. Gradient Descent, Wikipedia https://en.wikipedia.org/wiki/Gradient_descent

3. Arduino Reference https://www.arduino.cc/en/Reference/HomePage

4. Pololu QTR https://www.pololu.com/product/961

5. Neural Network for Arduino http://robotics.hobbizine.com/arduinoann.html

6. Fritzing http://fritzing.org/learning/full_reference